Preemption Delay Analysis for Floating Non-Preemptive Region Scheduling

José Marinho, Vincent Nélis, Stefan M. Petters, Isabelle Puaut

Technical Report

Preemption Delay Analysis for Floating Non-Preemptive Region Scheduling

José Marinho

Vincent Nélis

Stefan M. Petters

Isabelle Puaut

HURRAY-TR-111202

Version:

Date: 12-05-2011

Preemption Delay Analysis for Floating Non-Preemptive Region Scheduling

José Marinho, Vincent Nélis, Stefan M. Petters, Isabelle Puaut

IPP-HURRAY!

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8340509

E-mail:

http://www.hurray.isep.ipp.pt

Abstract

In real-time systems, there are two distinct trends for scheduling task sets on unicore systems: non-preemptive and preemptive scheduling. Non-preemptive scheduling is obviously not subject to any preemption delay but its schedulability may be quite poor, whereas fully preemptive scheduling is subject to preemption delay, but benefits from a higher flexibility in the scheduling decisions. The time-delay involved by task preemptions is a major source of pessimism in the analysis of the task Worst-Case Execution Time (WCET) in real-time systems. Preemptive scheduling policies including non-preemptive regions are a hybrid solution between non-preemptive and fully preemptive scheduling paradigms, which enables to conjugate both world's benefits. In this paper, we exploit the connection between the progression of a task in its operations, and the knowledge of the preemption delays as a function of its progression. The pessimism in the preemption delay estimation is then reduced in comparison to state of the art methods, due to the increase in information available in the analysis.

Preemption Delay Analysis for Floating Non-Preemptive Region Scheduling

José Manuel Marinho*, Vincent Nélis*, Stefan M. Petters*, Isabelle Puaut†
*CISTER-ISEP Research Centre, Polytechnic Institute of Porto, Portugal
†University of Rennes 1, UEB, IRISA, Rennes, France
Email: {jmsm,nelis,smp}@isep.ipp.pt , Isabelle.Puaut@irisa.fr

***Abstract*—In real-time systems, there are two distinct trends for scheduling task sets on unicore systems: non-preemptive and preemptive scheduling. Non-preemptive scheduling is obviously not subject to any preemption delay but its schedulability may be quite poor, whereas fully preemptive scheduling is subject to preemption delay, but benefits from a higher flexibility in the scheduling decisions. The time-delay involved by task preemptions is a major source of pessimism in the analysis of the task Worst-Case Execution Time (WCET) in real-time systems. Preemptive scheduling policies including non-preemptive regions are a hybrid solution between non-preemptive and fully preemptive scheduling paradigms, which enables to conjugate both world's benefits. In this paper, we exploit the connection between the progression of a task in its operations, and the knowledge of the preemption delays as a function of its progression. The pessimism in the preemption delay estimation is then reduced in comparison to state of the art methods, due to the increase in information available in the analysis.**

I. Introduction

Nowadays processors are composed of several subsystems (such as caches, pipelines, transfer lookaside buffers, etc.) which display, at any time-instant, an associated "*state*". In the context of this work, what we understand by "state of a subsystem" is the history information enclosed in the subsystem, as well as its logical configuration, at a particular time-instant. For example, the state of a cache at a given time-instant can be seen as a snapshot of all the information stored in that cache at that instant. The objective of this state is to accelerate average-case execution times. All these processor subsystems quasi-continuously face state changes at run-time and we are concerned with state changes which affect the temporal behavior of the executing tasks. In particular, it is the case for task preemptions: when a task resumes its execution (after being preempted), for example, the cache(s) will display a state which is different from its state at the time the task got interrupted. If its state has been substantially altered during the time the task was pending, it is likely that it might be needed to reconstruct at least partially its working set after the the task resumes execution.

Reconstructing the subsystems' states is attached to time penalties in real processors, which may increase the execution time of a task by non-negligible amounts of time. In general purpose computing this effect is balanced by the usually huge performance gains by deploying such state carrying sub-systems and hence is in most cases ignored. In embedded real-time systems, where timeliness is an essential property of the system and is hence thoroughly analysed, these penalties need to be carefully evaluated to ensure that all deadlines are met.

In this work we will mainly focus on the cache-related preemption delay (CRPD), because this delay has the most important impact on the variation of the execution time of a preempted task [1]. Knowledge of preemption delays is crucial for the assessment of the timing behavior of task-sets when scheduled by a given scheduling policy.

Real-time scheduling policies may be broken into three broad categories, with respect to how preemptions are handled: *(a) Non-preemptive scheduling*, where task preemptions are *not* allowed, *(b) Fully-preemptive scheduling*, where the highest priority active task always gets hold of the processor as soon as it arrives in the system (by interrupting the current executing task if needed), and *(c) Limited preemptive scheduling*, a hybrid solution between non- and fully-preemptive scheduling. This latter category can be itself divided into two subcategories: *fixed non-preemptive region scheduling*, where preemption points are hard-coded in the task's code and preemptions are allowed only when the execution of a task reaches one of these preemption points, and *floating non-preemptive region scheduling*. In the latter one, whenever a higher priority task is released, the currently running task starts to execute in a non-preemptive region. The length of this non-preemptive region is constant and defined in static time for each task. When the duration of the non-preemptive region elapses, the regular priority relationship between tasks is applied and the task with the highest priority is dispatched onto the processor potentially collating several preemptions into a single point.

On the one hand the *floating* non-preemptive regions model is more flexible than the *fixed* one and does not require modifications in the applications. On the other hand it restricts the time-locations at which the preemptions may take place, which makes it more predictable than *fully*-preemptive scheduling policies. These policies thus provide the system designer with more information about how the system will behave and decrease the pessimism involved in the analysis. It is important to state that the schedulability of these restricted preemption policies dominate over the fully preemptive ones [2]. The theory devised onwards assumes the scheduling using *floating non-preemptive regions* and proposes a new approach to safely but more tightly bound the preemption delay suffered by a task when compared to the state-of-the-art.

II. Related Work and Contribution

CRPD estimation has been a subject of wide study. Several methods have been proposed that provide an off-line estimation based on static analysis, for this inter-task interference value. Even though the work was later refined we will only discuss initial work. Lee et al. presented one of the earliest works on CRPD estimation for instruction-caches [3] where the concept of useful cache blocks was introduced.

Computation of the CRPD in data caches has been proposed by Ramaprasad and Mueller [4]. Since the assumption used in [3], that the value of CRPD throughout a control flow graph's basic block would remain constant, no longer holds for data caches a different approach had to be devised.

Preemption delay estimation is of little value without its integration into the schedulability test of the systems. Since preemption delay is affected by all elements of the task-set several approaches exist to handle this situation. Scheduling analysis by Lee [3] is based on response-time analysis (RTA) by using the k highest values of preemption delay and incorporating that quantity into the response time of the task. Lee uses integer linear programming (ILP) to compute the preemption delay suffered by each task.

Busquets et al. also used RTA [5], but considered the maximum effect the preempted task may suffer by multiplying the number of preemptions with the maximum CRPD. While this is more pessimistic than Lee's approach, it removes the complex analysis of intersecting cache sets which for realistically sized programs suffers from heavy state explosion.

Also a less complex algorithm in comparison to Lee's resorting to RTA was presented by Petters and Färber [1]. Opposed to Busquets' approach Petters uses the knowledge of the maximum damage each preempting task may cause instead of only considering the worst-case preemption delay. The ILP problem is addressed by using an iterative algorithm.

Altmeyer et al. presents a summary of all of the literature so far relative to preemption delay on fully preemptive fixed task priority [6]. The authors also presented an enhancement to the available work by merging the approaches of Petters and Busquets in a safe way and considering jitter and the preemption delay suffered by the shared resource execution. A demand-bound function based procedure has been proposed by Ju et al. [7], while the general approach of computing the CRPD is similar to Lee's approach.

All of the presented preemption delay-aware schedulability tests are specific to fully preemptive scheduling and are much more pessimistic than the one presented in this work since they do not consider the evolution of the preemption delay with the program progression of the preempted task. Our approach differs from past work in the sense that it ties the preemption delay with program-execution progression, thus enabling less pessimism in the preemption delay estimation.

Restricting preemption points presents a viable way to address the problem of preemption delay. The mechanism of preemption deferral was first proposed by Burns et al. [8]. It has a number of advantages as has been pointed out in several works e.g. [9], [10]. In particular, Gang Yao et al. provide a comparison of all the available methods described so far in literature [10] regarding restricted preemptive scheduling using fixed task priority.

Bertogna and Baruah have devised a method to compute the size of the non-preemptive regions, for earliest deadline first (EDF) scheduling policy, using a demand-bound function based technique [2]. In this work the slack in the schedule depending on the length of the interval, assuming synchronous release of all the tasks, is computed. The method fits both the fixed non-preemptive region model and the floating one.

Several methods addressing the same issue in fixed task priority exist [11], [12]. A fixed priority scheduling method has been devised by Gang Yao et al. [11], where a maximum bound on the length of fixed non-preemptive regions is provided. In this situation the computed length of the fixed non-preemptive regions are generally larger than in previous work, as the last chunk of a task's execution is not subject to further preemptions. This enables a further reduction on the number of preemptions.

Marinho and Petters presented a method to increase, at run-time, the length of the preemption triggered floating non-preemptive regions for fixed task priority [12]. This method is taking advantage of off-line knowledge and on-line task release information to increase the length of the non-preemptive regions. This leads generally to a steep decrease on the preemptions suffered. Similar to previous work the preemption delay problem was not addressed in their work. Reducing the number of preemptions helps decreasing the pessimism added to the schedulability test.

The preemption delay estimation problem using fixed non-preemptive region scheduling was presented by Bertogna et al. [13]. In order to reduce CRPD, the usage of fixed non-preemptive areas of code is proposed. The preemption points are thus reduced to a small number of well defined points. In this way the maximum CRPD is decreased and overall system's response time is enhanced. This work has the limitation that it requires manipulation of the code of tasks and thus is not very amenable to system developers. In particular, it is not straightforward to take into account tasks with complex control flow graphs [13]. Additionally it can not be easily applied in situations where the task-sets are subject to run-time change, since the maximum allowed distance between preemption points is defined by the higher priority workload.

Our work addresses the computation of the preemption delay in systems using preemption triggered floating non-preemptive regions which was previously not covered in the literature.

III. System Model

The system consists of a task set $\tau = \{\tau_1, \ldots, \tau_n\}$ scheduled to run on a single core processor. Each task τ_i may generate a potentially infinite sequence of jobs, which are the entities that contend for the processor usage. For each task, we assume that we have an estimation of its worst-case execution time (WCET) denoted by C_i.

There is an inherent priority relation between the jobs which governs the contention for the processor. This contention will be treated in a limited preemption model, which means that preemptions are allowed but are subject to some restrictions. This work supports both fixed task priority [11] and EDF [2] with floating non-preemptive region scheduling policies.

A floating non-preemptive region starts when the highest priority job is executing on the processor and a higher priority job is released. We denote by Q_i the length of the non-preemptive regions of task τ_i. This means that once a floating non-preemptive region has started, it will last for Q_i time units unless the currently running job completes before. Therefore, the preemption points which lead to the worst-case cumulative preemption delay are subject to the constraint of being distanced by *at least* Q_i time units apart. The first preemption can only happen after the task τ_i has completed Q_i units of execution. In this situation a higher priority release occurred at the same exact moment at which τ_i started execution. It is likely that the first preemption will

occur after τ_i has progressed further than Q_i. The Q_i value is a characteristic of each task. If the currently running job has not yet finished after the Q_i time units elapse then the highest priority job in the ready queue preempts it. The determination of Q_i can be performed by following the approaches determined in Bertogna et al. [2] or Marinho and Petters [12] and is assumed given within this work.

When a preempted task (say τ_i) resumes its execution, its remaining execution time will eventually increase, in comparison to the situation in which it was not preempted. This effect is due to the loss of working set in the hardware state. Within this work we focus on the largest contributor which is the CRPD. We call this increase in the remaining execution time the preemption *delay* that the task τ_i has to account for. This delay is as high as the amount of information, useful for the remaining execution of τ_i, evicted during the preemption. The preemption delay varies during the execution of the job. Let illustrate that with a simple example. Suppose that a task starts its execution by loading from the memory an important amount of data. Then the task processes all these data in a short period of time and finally, it performs a long-time computation using only a small subset of the data. In this case, the maximum preemption delay during the beginning of the task will be high, since in the worst-case scenario all the loaded data might be evicted during a preemption, hence forcing the task to reload them at the return from preemption. Then, once the data have been processed, the maximum preemption delay falls drastically, since a preemption during the long-time computation can only force the task to reload the few data that it needs when resuming its execution.

Each task is then characterised by a task-specific preemption delay pattern. As jobs execute their preemption delay varies with their progression through their execution. We model this varying cost of every task τ_i using a preemption delay function f_i. As such, it displays, for any time-instant t where the function is defined, an *upper bound* on the preemption cost that the task would incur if it was preempted at time t. This function is only valid for the first preemption since it does not take into account the preemption delay that has to be paid in the post-preemption execution.

IV. Coupling Preemption Delay Cost With Execution Points

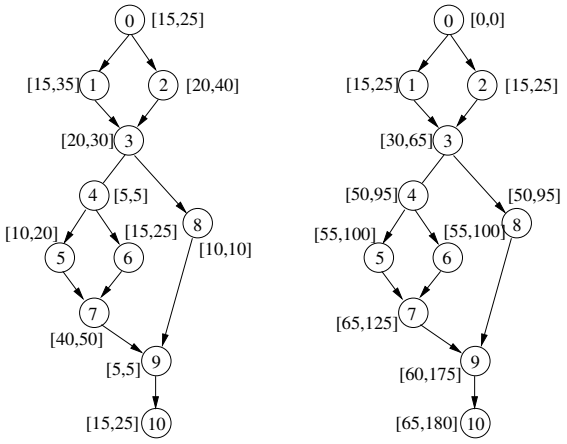
This section focuses on determining the initial preemption delay function f_i of each task τ_i. For that purpose, one first needs to obtain for every task represented by its control-flow graph (see Figure 1.a), the interval of time $[s_b^{\min}, e_b^{\max}]$ during which every basic block b might execute, considering the execution of τ_i in isolation.

Computing execution intervals on loop-free code requires to know for every basic block b its earliest and latest start offsets $s_b^{\min}$ and $s_b^{\max}$. This can be done by a breadth-first traversal of the CFG, applying to every traversed basic block b the following formulas:

$$s_{\text{entry}}^{\min} \stackrel{\text{def}}{=} s_{\text{entry}}^{\max} \stackrel{\text{def}}{=} 0 \tag{1}$$

$$s_b^{\min} \stackrel{\text{def}}{=} \min_{x \in \text{pred}(b)} (s_x^{\min} + e_x^{\min}) \tag{2}$$

$$s_b^{max} \stackrel{\text{def}}{=} \max_{x \in \text{pred}(b)} (s_x^{\max} + e_x^{\max}) \tag{3}$$



(a). CFG with execution time intervals per basic block

(b). CFG with computed earliest and latest start offsets per basic block

Fig. 1. Example of CFG for loop-free code. The CFG is composed of several basic blocks (0..10) connected by directed edges that represent jumps in the code. Each basic block is a set of sequential instructions delimited by a jump. In the left part, intervals $[c_b^{\min}, c_b^{\max}]$ represent the minimum and maximum execution times of basic block b. In the right part, intervals $[s_b^{\min}, s_b^{\max}]$ represent the earliest and latest start time of every basic block b.

with $\text{pred}(b)$ the direct predecessor(s) of a basic block b in the CFG, and entry the task entry basic block. In the formulas, $e_x^{\min}$ (resp. $e_x^{\max}$) represents the minimum (resp. maximum) execution time of basic block x; such values can be produced by standard WCET estimation tools. Figure 1.b) shows for every basic block its earliest and latest start offset after applying the above formulas. Then, the time interval within which every basic block b may execute is $[s_b^{\min}, s_b^{\min} + e_b^{\max}]$.

This method can be extended in a straightforward manner to programs with natural loops. The algorithm presented above can be applied to every loop individually, starting with the innermost. A loop can then be considered as a single node with known earliest and latest start offsets when analyzing the outer loop of the whole program. Similarly, tasks containing function calls can be analyzed provided that their call graph is acyclic by first analyzing the leaves in the call graph.

Knowing the possible execution interval $[s_b^{\min}, e_b^{\max}]$ of every basic block b, the set of basic blocks that might execute at a given time instant t, noted $\text{BB}(t)$ is known. For each basic block b in this set, state of the art methods like the one described in [3] is used to compute the maximum CRPD when preempting the task in basic block b, noted CRPD_b. More formally, function f_i can be defined as follows:

$$f_i(t) \stackrel{\text{def}}{=} \max_{\forall b \in \text{BB}(t)} (\text{CRPD}_b)$$

V. Determination of Preemption Delay Upper-bounds

As stated in Section III, a task will always execute non-preemptively for at least Q_i time units before a preemption occurs, unless it completes before the end of the non-preemptive region.

A naive thought to upper-bound the cumulative preemption delay over a task's execution (say τ_i) might be to select from f_i the maximum number of points p_k (each distanced from every other by at least Q_i time units) such that the sum $\sum_{\forall p_k} f_i(p_k)$ is maximum. However, the simple example depicted in Figure 2 shows that this solution is not correct. As we can see, on the top

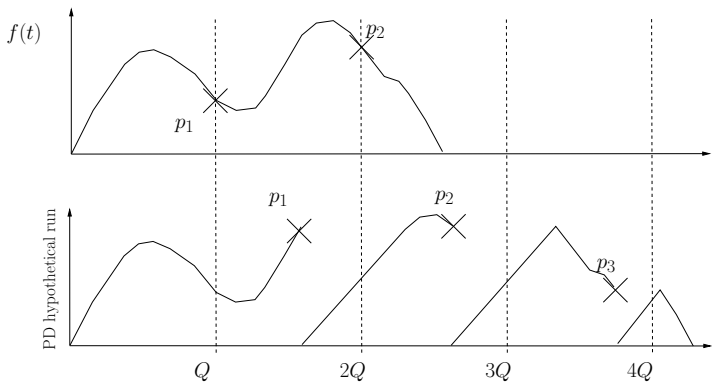


Fig. 2. Comparison Between Function f_i and the Run-time Preemption Delay Development

plot where f_i is depicted, there are at most two points that may be selected (since no three points could be distanced by at least Q_i time units in time). The bottom plot presents an hypothetical run of task τ_i, where the run-time preemption delay cost is presented. At run-time, since time is spent paying preemption delay after each preemption, more points can be selected (see the bottom plot), hence providing a higher cumulative preemption delay.

A pessimistic, but correct, solution to upper-bound the execution time C_i' of a task τ_i while taking into account all the possible preemption delays that τ_i might undergo during its execution, is simply to multiply the maximum number of preemptions that can occur during τ_i's execution (i.e., $\left\lfloor \frac{C_i}{Q_i} \right\rfloor$, this is discussed in more detail in [12]) by the maximum delay of one preemption (i.e., $\max_{t\in[0,C_i]} f_i(t)$). Given the increase in the WCET due to this cumulative overhead, the maximum number of preemptions that can occur eventually increases as well. Therefore, this computation has to be performed iteratively, in the style of the well-know task response-time computation, i.e., $C_i'^{(0)} = C_i$ and

$$C_i'^{(k)} = C_i + \left\lfloor \frac{C_i'^{(k-1)}}{Q_i} \right\rfloor \times \max_{t\in[0,C_i]} f_i(t) \tag{4}$$

The pessimism of this computation comes directly from the fact that it considers a constant cost for every possible preemption, and this constant cost is assumed to be the maximum possible cost. That is, this approach is not sensitive to the preemption cost pattern of the task. As it was claimed in the abstract, using this additional information (the tasks preemption cost pattern) enables us to derive a much more accurate upper-bound. This second technique is described in Algorithm 1, and a detailed explanation is provided below.

Description of Algorithm 1. Initially we will explain the intuition of the approach on Figure 3 before presenting the actual algorithm. In Figure 3 the gray curve is the f_i function. Suppose that prog is the current progression in the task execution. Considering the next preemption point, the approach is looking for the lower bound on the progression which will be achieved within the next Q_i time units in any preemption scenario. For this, function f_i is investigated from the current prog to $\mathrm{prog}+Q_i$. On the ordinate also at length Q_i a line $D(x,t)$ is drawn to $\mathrm{prog}+Q_i$. The point $p_\cap$ where f first crosses $D(x,t)$ limits the range of values which need to be considered. A preemption past this value would lead to a situation where this point would again

Algorithm 1: Upper-Bound the Preemption Delay

Input : $f_i()$: preemption delay function of task τ_i
Q_i: length of the non-preemptive region
Output: total_delay: cumulative preemption delay suffered by τ_i

```
1  prog ← 0 ;
2  total_delay ← 0;
3  delay_max ← 0 ;
4  p_next ← Q_i ;
   /* While the next progression is not beyond C_i  */
5  while p_next < C_i do
       /* Update time, progression and delay       */
6      prog ← p_next ;
       /* Compute the next progression step and the
          next delay to account for                */
7      p_∩ ← min{p_x} such that
8              p_x ∈ [prog^(k), prog^(k) +Q_i]
9              and  f_i(p_x) = −p_x + prog^(k) +Q_i} ;
10     if p_∩ = null then p_∩ ← prog +Q_i;
11     p_max ← arg max_{p_x∈[prog,p_∩]}{f_i(p_x)};
12     delay_max ← f_i(p_max);
13     p_next ← prog +Q_i − delay_max;
14     total_delay ← total_delay + delay_max ;
15 return total_delay ;
```

be considered in a subsequent iteration, since then prog would not pass this point in the current iteration. Within the interval, $\mathrm{delay}_{\max}$ is determined. That means in an interval Q_i under any preemption scenario at least $Q_i - \mathrm{delay}_{\max}$ progress in program execution will be achieved. It is a conservative bound as a later preemption means that also the non preemptible region will only start then. This point $\mathrm{prog}+Q_i - \mathrm{delay}_{\max}$ will serve as new starting point.

Returning to the Algorithm 1: Lines 1–4 initialise the variables. The variable prog memorizes the current progression in the task's operations while $\mathrm{total\_delay}$ records the cumulative preemption delay accounted for up to the current progression. As the task τ_i executes, it accounts for progressing in its execution (and the variable prog is increased) and for the preemption delay (which updates the variable $\mathrm{total\_delay}$). The algorithm is iterative, and at each iteration the variables $\mathrm{delay}_{\max}$ and p_{next} (lines 3 and 4) are the preemption delay taking place only in the current iteration and the next progression point in τ_i's execution at which the next iteration will start, respectively. Lines 1–4 can be seen as the first iteration of the algorithm. $\mathrm{delay}_{\max}$ is set to 0 and p_{next} to Q_i,

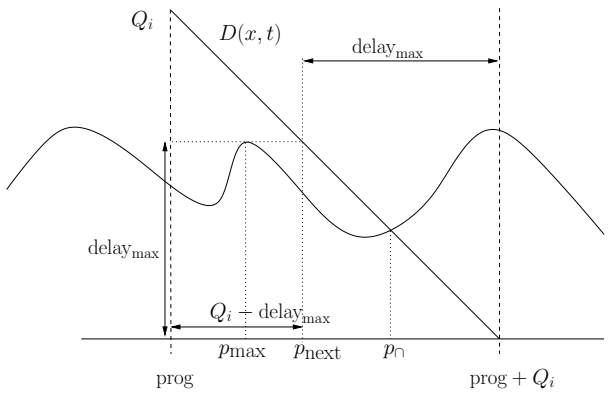


Fig. 3. Algorithm iteration sketch

because no preemption can occur during the first Q_i time units of τ_i's execution.

The algorithm starts iterating at line 5, and it iterates as long as the next computed progression point p_{next} does not fall beyond τ_i's execution boundary. Line 6 shifts the current progression point of τ_i to the computed value p_{next}. Then, lines 12 and 13 compute the next progression point p_{next} and the maximum delay that τ_i could suffer while progressing in its operations from its current progression point to p_{next}. Finally, line 14 adds this maximum delay to the current cumulative delay accounted so far.

In the following Theorem 1, we prove that the value returned by Algorithm 1 is an upper-bound on the cumulative preemption delay that the given task τ_i might suffer during its execution. This implies that the WCET of τ_i (while taking into account all the possible preemption delays that τ_i might suffer during its execution) is given by

$$C_i' \stackrel{\text{def}}{=} C_i + \text{total\_delay} \tag{5}$$

where total_delay is the value returned by Algorithm 1.

Theorem 1. *Algorithm 1 returns an upper-bound on the preemption delay that a given task τ_i can suffer during the execution of any of its jobs.*

Proof: Algorithm 1 computes the maximum cumulative preemption delay iteratively, by progressing step by step through the execution of the task τ_i. Hereafter, we use the notation $\text{prog}^{(k)}$ to denote the progression through τ_i's execution at the beginning of the k^{th} iteration of the algorithm. Similarly, $\text{total\_delay}^{(k)}$ will be used to denote the cumulative preemption delay that τ_i has suffered until it reached a progression of $\text{prog}^{(k)}$. In this proof, we show that at each iteration $k > 0$, $\text{total\_delay}^{(k)}$ actually provides an *upper-bound* on the cumulative preemption delay that τ_i might suffer before reaching a progression of $\text{prog}^{(k)}$ in its execution. The proof is made by induction.

Basic step. Algorithm 1 first considers that τ_i progresses by Q_i time units in its execution without suffering any preemption delay (since it cannot get preempted during these first Q_i time units). We consider this first step as the first iteration of the algorithm. That is, straightforwardly, $\text{total\_delay}^{(1)} = 0$ is an upper (and even exact) bound on the cumulative preemption delay that τ_i may suffer before reaching a progression of Q_i time units in its execution.

Induction step. We assume (by induction) that $\text{total\_delay}^{(k)}$, $k > 1$, is an upper-bound on the cumulative preemption delay that τ_i might suffer before reaching a progression of $\text{prog}^{(k)}$ time units in its execution.

During the k^{th} iteration, Algorithm 1 computes $\text{prog}^{(k+1)}$ and $\text{total\_delay}^{(k+1)}$ as follows:

$$\text{prog}^{(k+1)} = \text{prog}^{(k)} + Q_i - \text{delay}_{\max} \tag{6}$$

$$\text{total\_delay}^{(k+1)} = \text{total\_delay}^{(k)} + \text{delay}_{\max} \tag{7}$$

where

$$\text{delay}_{\max} = f_i(p_{\max}) \tag{8}$$

$$p_{\max} = \underset{p_x \in [\text{prog}^{(k)}, p_\cap]}{\arg\max} \{f_i(p_x)\} \tag{9}$$

$$p_\cap = \min\{p_x\} \text{ such that } p_x \in [\text{prog}^{(k)}, \text{prog}^{(k)} + Q_i] \text{ and } f_i(p_x) = -p_x + \text{prog}^{(k)} + Q_i \tag{10}$$

Equations 6 and 7 can be interpreted as follows. During the k^{th} iteration, Algorithm 1 assumes that τ_i executes for Q_i time units during which τ_i progresses by $Q_i - \text{delay}_{\max}$ units of time in its execution and suffers from a delay of $\text{delay}_{\max}$; The algorithm assumes that τ_i gets preempted when its progression reaches $p_{\max}$ given by Equation 9. Below we show that choosing any other preemption point $p_{\text{other}} \neq p_{\max}$ would ultimately[1] result in a cumulative preemption delay lower than the one returned by Algorithm 1, thus showing that the value returned by Algorithm 1 is an upper-bound. Two cases may arise: $p_{\text{other}} > p_{\text{next}}$ or $p_{\text{other}} \leq p_{\text{next}}$.

Case 1: $p_{\text{other}} > p_{\text{next}}$. This means that τ_i progresses in its execution until it reaches p_{next} *without being preempted*, i.e., from a progression of $\text{prog}^{(k)}$, τ_i reaches a progression of p_{next} by being executed only for $(p_{\text{next}} - \text{prog}^{(k)})$ time units, and with an unchanged cumulative preemption delay of $\text{total\_delay}^{(k)}$. On the other hand, in the execution scenario built by Algorithm 1, τ_i's execution reaches a progression of $\text{prog}^{(k+1)} = p_{\text{next}}$ by being executed for Q_i time units, and with a cumulative preemption delay of $\text{total\_delay}^{(k+1)} = \text{total\_delay}^{(k)} + \text{delay}_{\max} \geq \text{total\_delay}^{(k)}$. In other words, Algorithm 1 manages to progress slower in τ_i's execution while suffering from a greater preemption delay. Furthermore, p_{other} is still a candidate preemption point for a further iteration of Algorithm 1.

Case 2. $p_{\text{other}} \leq p_{\text{next}}$. *After executing τ_i for Q_i time units*, we have that

1) the delay of the preemption that occurs when τ_i's progression reaches p_{other} has been totally accounted for (since $p_{\text{other}} < p_{\text{next}} \leq p_\cap$).
2) the progression of τ_i in this scenario becomes

$$\begin{aligned} \text{prog}_{\text{other}} &= \text{prog}^{(k)} + Q_i - f_i(p_{\text{other}}) \\ &\geq \text{prog}^{(k)} + Q_i - f_i(p_{\max}) \\ &\geq \text{prog}^{(k+1)} \end{aligned} \tag{11}$$

3) the cumulative preemption delay becomes

$$\begin{aligned} \text{total\_delay}_{\text{other}} &= \text{total\_delay}^{(k)} + f_i(p_{\text{other}}) \\ &\leq \text{total\_delay}^{(k)} + f_i(p_{\max}) \\ &\leq \text{total\_delay}^{(k+1)} \end{aligned} \tag{12}$$

Thus, after executing τ_i for Q_i time units Algorithm 1 progressed less in the execution of τ_i (Inequality 11) while suffering from a higher preemption delay (Inequality 12). As a consequence of Cases 1 and 2, it holds at each iteration of Algorithm 1 that choosing to preempt the task when it reaches a progression of $p_{\max}$ ultimately leads to an upper-bound on its cumulative preemption delay. ■

[1] when τ_i's execution will be completed

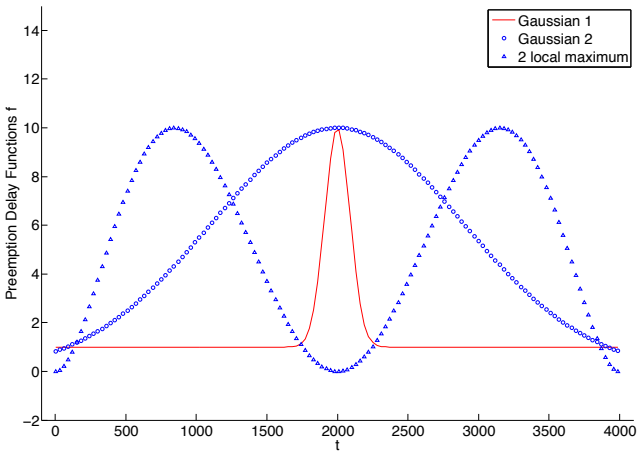


Fig. 4. Synthetic Benchmark Functions

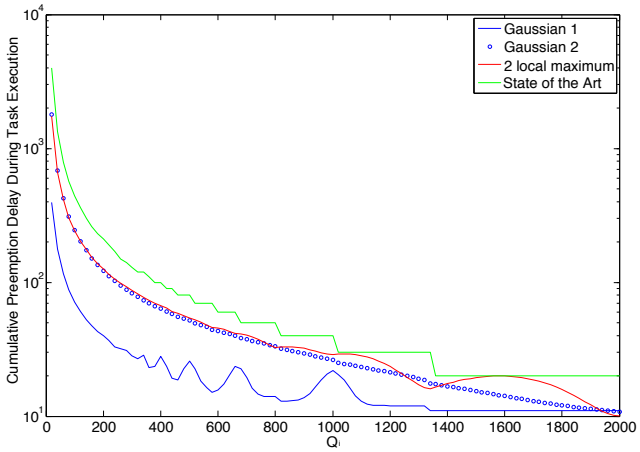


Fig. 5. Benchmark Results

VI. Evaluation and Discussion

Three synthetic f_i functions have been created in order to compare the performance of the proposed preemption delay estimation with the state-of-the-art method using the procedure described by Equation 4. The three functions used are two bell shaped functions, the first one with $\sigma^2 = 300$ and $\mu = 2000$ and a vertical offset of 10 units and the second one with a bigger variance, $\sigma^2 = 3000$, the same average and no offset. Finally a function with two local maxima separated in time is used. All functions have maximum value of 10 units, and have $C = 4000$. We vary the Q_i value so having a fixed C still paints a generic picture of how the methods behave.

The synthetic functions used represent distinct generic memory usage patterns from tasks. The sole purpose of the set of functions provided is to validate the method proposed in this paper. Having generic patterns, rather than function f_i extracted from a set of real benchmarks (which would present more complex patterns), enables for a more clear evaluation of Algorithm 1 performance. These functions are portrayed in Figure 4.

The proposed algorithm is shown empirically, in Figure 5, to provide a considerably less pessimistic upper-bound on the preemption delay value for a task, specially for smaller values of Q_i. Since the state of the art method purely relies on Q_i, C_i and the maximum preemption delay of f_i then its preemption delay estimate is strictly the same for all the benchmark functions, since they all have the same C and maximum value. The preemption delay axis in Figure 5 is in logarithmic scale so that the differences between the state of the art and the proposed algorithm are more easily observed across the entire Q_i spectrum.

There are fluctuations in the results which are analysis artifacts and imply that the analysis is pessimistic. In some cases increasing the Q_i results in bigger preemption delay. This is caused as the analysis checks for the preemption delay in the window of prog and t_A, but conservatively considers the actual preemption to occur at prog. An actual preemption at $p_{\max}$ in physical terms would initiate a new window of length Q_i to start only at that $p_{\max}$ instead of prog , thus the method, as is proven in Theorem 1, provides an upper bound for the preemption delay in any conceivable real task execution scenario.

VII. Conclusion and Future Work

In this work we have proposed a new algorithm to compute an upper-bound on the preemption delay suffered by a task which executes in a system scheduled with preemption triggered floating non-preemptive regions. This algorithm has been shown to dominate over the state-of-the-art method. The method is easy to implement with small overhead and builds up on existing static-analysis methods.

As of future work we intend to tighten our result by (i) discarding less information during the computation of function $f_i(t)$ and (ii) reducing the number of preemptions (i.e., the number of iterations) considered in Algorithm 1 – it is indeed impossible for a task to get preempted every Q_i time units as assumed by Algorithm 1 unless the periods of the other tasks enable such a preemption scenario.

References

[1] S. M. Petters and G. Färber, "Scheduling analysis with respect to hardware related preemption delay," in *Workshop on Real-Time Embedded Systems*, London, UK, Dec 2001.

[2] M. Bertogna and S. Baruah, "Limited preemption edf scheduling of sporadic task systems," *IEEE Transactions on Industrial Informatics*, vol. 6, no. 4, Nov 2010.

[3] C.-G. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim, "Analysis of cache-related preemption delay in fixed-priority preemptive scheduling," *IEEE Transactions on Computers*, vol. 47, no. 6, 1998.

[4] H. Ramaprasad and F. Mueller, "Bounding preemption delay within data cache reference patterns for real-time tasks," in *12th RTAS*, Apr 2006.

[5] J. Busquets-Mataix, J. Serrano, R. Ors, P. Gil, and A. Wellings, "Adding instruction cache effect to schedulability analysis of preemptive real-time systems," in *17th RTSS*, Jun 1996.

[6] S. Altmeyer, R. I. Davis, and C. Maiza, "Pre-emption cost aware response time analysis for fixed priority pre-emptive systems," in *32nd RTSS*, Nov 2011.

[7] L. Ju, S. Chakraborty, and A. Roychoudhury, "Accounting for cache-related preemption delay in dynamic priority schedulability analysis," in *44th DATE*, Apr 2007.

[8] A. Burns, "Preemptive priority-based scheduling: an appropriate engineering approach," in *Advances in real-time systems*, S. H. Son, Ed. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1995.

[9] Y. Wang and M. Saksena, "Scheduling fixed-priority tasks with preemption threshold," in *6th RTCSA*, 1999.

[10] G. Yao, G. Buttazzo, and M. Bertogna, "Comparative evaluation of limited preemptive methods," in *15th ETFA*, Sep 2010.

[11] ——, "Feasibility analysis under fixed priority scheduling with limited preemptions," *Journal Real-Time Systems*, vol. 47, no. 3, 2011.

[12] J. Marinho and S. M. Petters, "Job phasing aware preemption deferral," *International Conference on Embedded and Ubiquitous Computing 2011*, Oct 2011.

[13] M. Bertogna, O. Xhani, M. Marinoni, F. Esposito, and G. Buttazzo, "Optimal selection of preemption points to minimize preemption overhead," in *23th RTSS*, Jun 2011.